Visual querying and analysis of large software repositories

Voinea, Lucian; Telea, Alexandru

Published in:
Empirical software engineering

DOI:
10.1007/s10664-008-9068-6

Publication date:
2009

Empir Software Eng (2009) 14:316–340
DOI 10.1007/s10664-008-9068-6

Visual querying and analysis of large software repositories

Lucian Voinea · Alexandru Telea

Published online: 10 May 2008

Editors: Prof. Hassan, Prof. Diehl and Prof. Gall

Abstract We present a software framework for mining software repositories. Our extensible framework enables the integration of data extraction from repositories with data analysis and interactive visualization. We demonstrate the applicability of the framework by presenting several case studies performed on industry-size software repositories. In each study we use the framework to give answers to one or several software engineering questions addressing a specific project. Next, we validate the answers by comparing them with existing project documentation, by interviewing domain experts and by detailed analyses of the source code. The results show that our framework can be used both for supporting case studies on mining software repository techniques and for building end-user tools for software maintenance support.

Keywords Software visualization · Evolution visualization · Repository mining

1 Introduction

Software configuration management (SCM) systems are widely accepted instruments for managing large software development projects containing millions of lines of code spanning thousands of files developed by hundreds of people over many years. SCMs maintain a history of changes in both the structure and contents of the managed project. This information is suitable for empirical studies on software evolution (Ball et al. 1997; Bennett et al. 1999; Greenwood et al. 1998).

L. Voinea (✉)
SolidSource BV, Grassavanne 14, 5658 EV Eindhoven, The Netherlands
e-mail: lucian.voinea@solidsource.nl

A. Telea
Institute of Mathematics and Computer Science, University of Groningen,
Nijenborgh 9, 9747 AG Groningen, The Netherlands
e-mail: a.c.telea@rug.nl

Many SCM systems exist on the market and are used in the daily practice in the software industry. Among the most widespread such systems we mention CVS, Subversion, Visual SourceSafe, RCS, CM Synergy, and ClearCase. The Concurrent Versions System (CVS) and Subversion are, in particular, very popular in the area of open source software projects. Many CVS and Subversion repositories covering long evolution periods, e.g., 5–10 years, are freely available for analysis. These repositories are, hence, interesting options for research on software evolution.

Most SCM systems, including CVS and Subversion, are nevertheless primarily designed to support the task of archiving software and maintaining code consistency during development. The main operations they provide are check-in, check-out, and a limited amount of functionality for navigating the intermediate versions that a software system has during its evolution. They offer no functionality that enables users to get data overviews easily. Such overviews are essential in case the questions asked involve *many* files or file versions, rather than a particular file, or address facts at a higher abstraction level than the file level. Most questions concerning spotting trends in software evolution are of this type. This leads us to a major challenge of software evolution research based on SCMs, namely how to tackle data size and complexity. Raw repository information is too large and provides, when directly displayed, only limited insight into the evolution of a software project. Extra analysis is needed to process such data and extract relevant evolution features.

Visual tools, added atop of basic SCM systems, are a recent advance in the field. By using dense pixel techniques, information on hundreds of versions of hundreds of files can be displayed on a single screen overview. Furthermore, interesting evolution patterns can be identified by directly looking at the visualization. Tuning various graphics parameters such as layout, texture, color, and shading yields different visualizations, thereby addressing different user questions.

However promising, a fundamental question remains: how valid are the answers produced by visualization tools, when compared to ground truth as known by domain experts? A related question is: can visualization tools be used to answer non-trivial questions on industry-size repositories? Although preliminary evidence suggests that such tools enhance the users' analysis powers and can provide accurate insight, there is still a high demand for a stronger validation of the accuracy of the obtained insight on complex, real-life code repositories which are unfamiliar to the persons using the tools. In this paper, we address the above question by a number of empirical studies, as follows. We first introduce an extensible framework for SCM data extraction and analysis. Then we customize this framework to support developers in answering a number of concrete questions on CVS and Subversion software repositories ($Q1 \cdots Q5$):

Q1: What is the contribution style in a given project?
Q2: Who are the main developers in charge?
Q3: What are the high-level building blocks of a software system?
Q4: How maintainable is a given project?
Q5: What is the maintenance risk of a given developer leaving a given project?

Next, we use our framework to perform several empirical studies on real-life software repositories. In these studies, one or several questions on a specific software repository ate to be answered by a person who is not the main code developer. We validate the obtained answers by comparing them with existing project doc-

umentation, by interviewing software developers and by detailed analyses of the source code itself. Finally, we use this information to draw several conclusions concerning the effectiveness of visual repository analysis in supporting software evolution assessments.

The structure of this paper is as follows. In Section 2 we review existing repository data extraction methods and software evolution analysis techniques. Section 3 introduces our customizable framework for mining software repositories: Section 3.1 presents a flexible data interface with software repositories; Section 3.2 describes a clustering technique for detecting logical coupling of files based on evolution similarity; Section 3.3 describes a visual back-end for evolution assessment. Next, in Sections 4 to 7 we demonstrate the applicability of our framework with four case studies performed on several large-scale, real-world repositories. The specific planning, operation, analysis, result interpretation and result validation are presented for each case. Section 8 discusses several aspects related to the choice of the case studies and their prototypical value. Section 9 summarizes our contribution and outlines open issues for future research.

2 Background

The huge potential of the data stored in SCMs for empirical studies on software evolution has been recently acknowledged. The growth in popularity and use of SCM systems, e.g., the open source (CVS, http://www.nongnu.org/cvs/) and Subversion (SVN, http://subversion.tigris.org), opened new ways for project accounting, auditing, and understanding. Existing efforts to support these developments can be grouped in three directions: data extraction, data mining, and data visualization.

Data extraction is a less detailed, yet very important, aspect of software evolution analysis. Subversion offers a standard API for accessing the software repository. However, Subversion is a relatively new SCM system. It appeared in June 2000 and it was first managed using CVS. This is reflected in the existence of fewer large repositories freely available for investigations. A large part of the open source community still uses CVS as their primary SCM system. Hence, a large part of research in software evolution consider data from CVS repositories, e.g., (Burch et al. 2005; Fischer et al. 2003; Gall et al. 2003; German et al. 2004; Lopez-Fernandez et al. 2004; Voinea and Telea 2006a; Voinea et al. 2005; Ying et al. 2004; Zimmermann et al. 2004). Yet, a standard framework for CVS data extraction does not exist so far. To build such a framework, two main challenges must be addressed: evolution data retrieval and CVS output parsing. The huge data amount in repositories is usually available over the Internet. On-the-fly retrieval is ill suited for interactive (visual) assessment, given the sheer data size. Storing data locally requires long acquisition times, large storage space, and consistency checks. Next, CVS output is badly suited for machine reading. Many CVS systems use ambiguous or nonstandard output formats (e.g., file names without quotes but including spaces, which compromise consistency in space separated records). Attempts to address these problems exist, but are incomplete. Libraries exist that offer an application interface (API) to CVS, e.g., `javacvs` (for Java programmers) or `libcvs` (for Perl programmers). However, `javacvs` is basically undocumented, and hence of little use. `libcvs` handles only local repositories. The Eclipse environment offers a CVS client but not an API.

The Bonsai project (Bonsai, http://www.mozilla.org/projects/bonsai/) offers tools to populate a database with evolution data obtained from CVS repositories. However, these tools are mainly meant as a web data access package and are little documented. The best supported effort for CVS data acquisition so far is the `NetBeans.javacvs` package (http://javacvs.netbeans.org) for Java programmers (not to be mistaken as `javacvs`). This package has a reasonably documented API offering most of the functionality of the standard CVS client by parsing its output into API-level data structures. However, the usefulness of this library depends on its ability to support nonstandard formats. Its main drawback is the difficulty to adapt it to a specific nonstandard situation.

Data mining focuses on extracting and processing relevant information from SCM systems. SCM systems have not been designed to support empirical studies, so they often lack direct access to high-level, aggregated evolution information. This is distilled from the "raw" stored data by data mining tools. In the following, we review a number of the prominent results in this field. Fischer et al. (2003) extend the SCM evolution data with information on file merge points. Gall et al. (2003) and German and Mockus (2003) use transaction recovery methods based on fixed time windows. Zimmermann et al. (2004) extended this work with sliding windows and facts mined from commit e-mails. Ball analyzes cohesion of classes using a mined probability of classes being modified together (Ball et al. 1997). Bieman et al. (2003) and Gall et al. (2003) also mine relations between classes based on change similarities. Ying et al. (2004) and Zimmermann and Weisgerber (2004), Zimmermann et al. (2004), address relations between finer-grained artifacts, e.g., functions. Lopez-Fernandez et al. (2004) apply general social network analysis methods on SCM data to assess the similarity and development process of large projects. Overall, the above methods provide several types of distilled facts from the raw data, thereby trying to answer questions such as: which software entities evolved together, which are the high and low activity areas of a project, how is the developers network structured, and how do a number of quality metrics, such as maintainability or modularity, change during a project's evolution. During data mining, a data size reduction often occurs, as only the metrics and facts relevant to the actual question are examined further. However useful, this approach can discard important facts which are not directly reflected by the mining process, but which can give useful collateral insight.

Data visualization takes a different path than data mining, focusing on making the large amount of evolution information available to users in an effective way. Visualization methods make few assumptions on the data - the goal is to let users discover patterns and trends rather than coding these in the mining process. SeeSoft (Eick et al. 1992), one of the earliest works in this direction, is a line-based code visualization tool which uses color to show code snippets matching given modification requests. Augur (Froehlich and Dourish 2004) visually combines project artifact and activity data at a given moment. Xia (Wu et al. 2004b) uses treemap layouts for software structure, colored to show evolution metrics, e.g., time and author of last commit and number of changes. Such tools successfully show the structure of software systems and the change dependencies at given moments. Yet, they do not show code attributes and structure changes made throughout an entire project. A first step towards getting insight into several versions, UNIX's `gdiff` and Windows' WinDiff tools display code differences between two versions of a file by showing line insertions, deletions, and edits computed by the `diff` tool. Still, such tools cannot

show the evolution of thousands of files and hundreds of versions. Collberg et al. solve this for medium-size projects by depicting the evolution of software structures and mechanisms as a sequence of graphs (Collberg et al. 2003). Lanza (2001) depicts the evolution of object-oriented software systems at class level. Wu et al. (2004a) visualize the evolution of entire projects at file level and emphasize the evolution moments. Voinea and Telea propose a set of tools that visualize the entire evolution of software at file (Voinea et al. 2005) and project level (Voinea and Telea 2006a) using dynamic layouts and multivariate visualization techniques. The approaches of Lanza, Wu et al., and Voinea and Telea scale well on large software projects. Burch et al. (2005) propose a framework for visual mining of both evolution and structure at various levels of detail. SoftChange (German et al. 2004) is an attempt for a coherent environment to support the comparison of Open Source projects, targeting CVS, project mailing lists, and bug report databases. SoftChange focuses mainly on data extraction and analysis, aiming to be a generic foundation for building evolution visualization tools.

In spite of numerous attempts to tackle challenges in each of the above mentioned directions, there exist only a few tools that combine data extraction, data mining, and data visualization functionality in a single environment (Cubranic et al. 2005; German et al. 2004). Tool integration is absolutely essential for getting them accepted by engineers during software development and maintenance, and also for organizing empirical studies on software evolution. Many existing tools, however, are quite monolithic and do not allow easy integration of new extraction, mining or visualization components. To address this issue, we present next a new approach towards an extensible framework.

3 An Extensible Framework for Mining Software Repositories

The purpose of our extensible framework for repository mining is twofold. First, it provides a foundation for empirical research on software evolution, both in terms of software tools and methodology. Secondly, it leverages the power of interactive visualization techniques in the analysis of large amounts of interesting and potentially useful information stored in software repositories. Ultimately, we would like to use this framework to provide end users with a complete software evolution analysis chain. We present next the key design elements of our framework and the components it integrates so far.

3.1 Data Extraction

As already explained in Section 2, repository data extraction is a serious practical problem for building analysis and visualization tools. Most repository access protocols, such as implemented in the CVS and Subversion versioning systems, implement only the basic access functions needed to support the archiving process. In the case of CVS, navigation commands also do not have a machine-readable output. When parsing CVS output, one usually searches for a parser that copes with the output format at hand and tries to add it to the experimental setup.

We propose next an approach towards repository data access that simplifies the process using a data acquisition mediator (see Fig. 1).

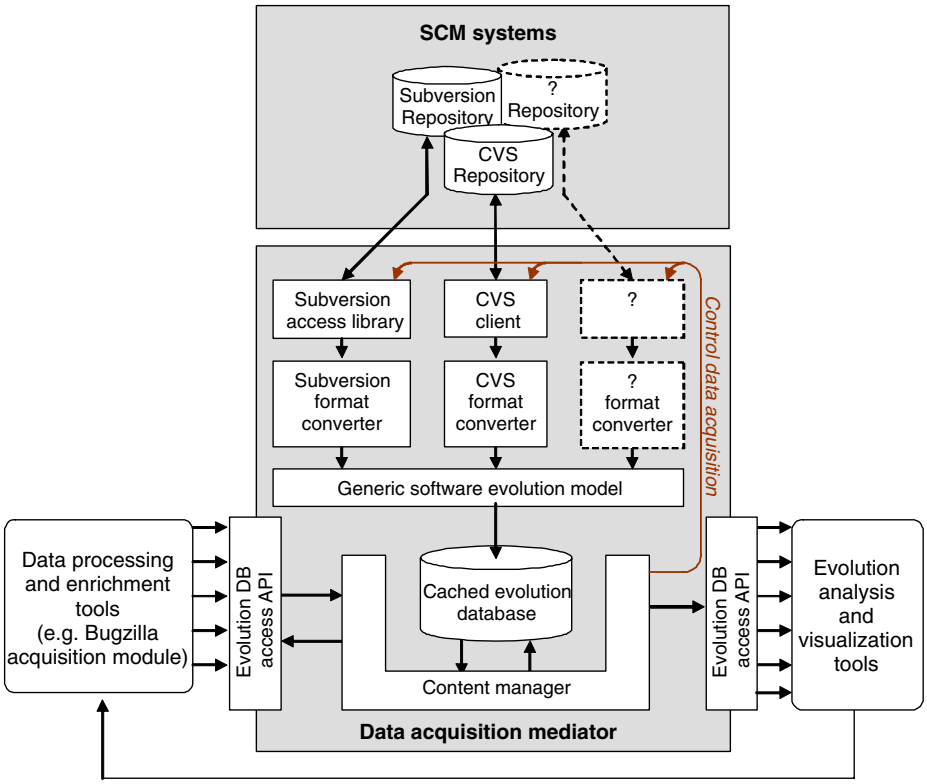

Fig. 1 Architecture of an extensible framework for visual software repository analysis

The mediator is an instrumental part of our framework. It forms an easy-to-customize layer between repositories, data acquisition, processing, and analysis tools. When format inconsistencies occur between the CVS output and a parser, we do not need a new CVS data acquisition tool. Instead, we adapt the mediator with a simple rule to transform the new format into the one accepted by the tool. While this does not completely remove the problems of inconsistent output formatting, it is a flexible way to solve problems without removing the preferred data acquisition tool. We developed an open source, easy to customize mediator, in a simple to use programming language: Python. The mediator consists of a set of scripts which act as data filters, i.e. read the raw data produced by the SCM tool (e.g. CVS client), apply data and format conversion, and output a filtered data stream. Filters can be connected in a dataflow-like manner to yield the desired data acquisition tool. Several filter pipelines can be added to work in parallel and have their outputs merged at the end. This enables alternative parsing strategies, e.g. in case one does not know beforehand how to detect the data format used.

A second function of the mediator is to offer selective access to CVS repositories by retrieving only information about a desired folder or file, as demanded by the data analysis or visualization tool further in the pipeline. The mediator also caches the retrieved information locally, using a custom developed database. This design

lets one transparently control the trade-off between latency, bandwidth and storage space in the data acquisition step as desired.

We have been able to test our mediator-based data extraction as part of several software repository mining challenges at the MSR'06, SoftVis'06, and Vissoft'07 scientific events (Voinea and Telea 2006b, c). Getting to the facts in the repository was one of the main time-consuming, tedious, tasks for most participants. The mediator solution, complemented by a small amount of scripting, provided us with a means to quickly get the facts and save more time for the actual visual analysis part.

3.2 Mining Evolutionary Coupling

Raw repository data is too large and low-level to provide insight into the evolution of software projects. Extra analysis is needed to extract relevant evolution aspects. An important analysis use-case is to identify artifacts that have similar evolution. Several approaches exist for this task (Bieman et al. 2003; Gall et al. 2003; Ying et al. 2004; Zimmermann et al. 2004). All these approaches use similarity measures (also known as evolutionary coupling measures) based on recovered repository transactions, i.e., sets of files committed by a user at some moment. The assumption is that related files have a similar evolution pattern, and thus their revisions will often share the same commit transaction. This information about correlated files is used to predict future changes in the analyzed system, from the perspective of a given artifact.

We offer a more general approach that does not take transactions into account but pure commit moments, when searching for similar files. We believe transaction-based similarity measures fail to correlate files that are developed by different authors, have different comments attached, and yet are still highly coupled. To handle such cases, we propose a similarity measure using the time distance between commit moments. If $S_1 = \{t_i | i = 1..N\}$ are the commit moments for a file F_1 and $S_2 = \{t_j | j = 1..M\}$ the commit moments for F_2, we define the similarity between F_1 and F_2 as the symmetric sum:

$$\Phi(F_1, F_2) = \sum_{i=1}^{N} \frac{1}{\sqrt{\max(\min_{t_j \in S_2} |t_i - t_j|, k)} + l} + \sum_{j=1}^{M} \frac{1}{\sqrt{\max(\min_{t_i \in S_1} |t_j - t_i|, k)} + l} \quad (1)$$

where k and l are parameters intended to reduce the influence of completely unrelated events on the similarity measure (k), and to limit the impact of isolated, yet very similar, events (l). The square root attenuates the influence of the network latency on the transaction time recorded by the repository. Intuitively, the measure Φ considers, for each commit moment of F_1, the closest commit moment from F_2, weighted by the inverse time distance between the two moments. In practice, we obtained good results by assigning value 1 to l and 3,600 (seconds) to k. The case studies presented in Section 5 uses these values.

We next use this measure in an agglomerative clustering algorithm to group files that have a similar evolution. The clustering works in a bottom-up fashion, as follows. First, all individual files are placed in their own cluster. Next, the two most similar clusters, as given by the above similarity metric, are found and merged in a parent

cluster, which gets the two original clusters as children. The process is repeated until a single root cluster is obtained. To measure the similarity of non-leaf clusters, we use (1) on all commit moments of all files in the first, respectively second, cluster. This is equivalent to a full linkage clustering which is computationally more expensive than other techniques such as single linkage or k means (Everitt et al. 2001), but offers a context-independent, more stable, result. To give a practical estimation of the efficiency: Clustering a project containing around 850 versions of 2,000 files, using a C++ implementation of the evolutionary clustering, took just under 5 min on a 2.5 GHz Windows PC.[1]

After the clustering tree is computed, it can be used to obtain a *decomposition* of system evolution by selecting only those clusters that satisfy a given criterion. Users can interactively select the most appropriate decomposition in a visual way using, for example, the cluster map widget introduced by Voinea and Telea in (Voinea and Telea 2006d). We added this facility to our framework, using the data acquisition mediator introduced in Section 3.1 to decouple the calculation of the clustering tree (done in C++ by a standalone program) from the cluster selection widget (done in Python as part of our visualization).

3.3 Visualization

Visualization attempts to provide insight into large and complex evolution data by delegating pattern detection and correlation discovery to the human visual system. Visualization can also present data analysis results in an intuitive way. Visualization is a main ingredient of our software repository mining framework.

As a visualization back-end of our framework we integrated the CVSgrab tool (Voinea and Telea 2006a). In its first release (2004), CVSgrab was a tool for acquiring history information from CVS repositories. Since then, CVSgrab has evolved into a set of techniques for visualizing project evolution at file level.

A complete project, as present in an SCM repository, is rendered in CVSgrab as a set of horizontal strips (Fig. 2 left). Each strip shows a file's evolution in time, along the x axis. The file strips are stacked (ordered) along the y axis in different ways (e.g. by age, type, size, number of changes, evolutionary coupling), thereby covering different types of analyses. Color can show several attributes: file type, size, author, metrics computed on the file contents, the occurrence of a specific word in the file's commit log, or information from an associated bug database.

CVSgrab can also visualize the evolution of file ownership, i.e., how many files carry the signature of a given user at one or several moments (Fig. 2 right). This type of visualization is particularly useful when assessing the team-related risks of a software project (see Section 7).

[1]The entire project contains more than 850 versions, but we were only interested analyzing a subperiod of its entire evolution that covered these versions.

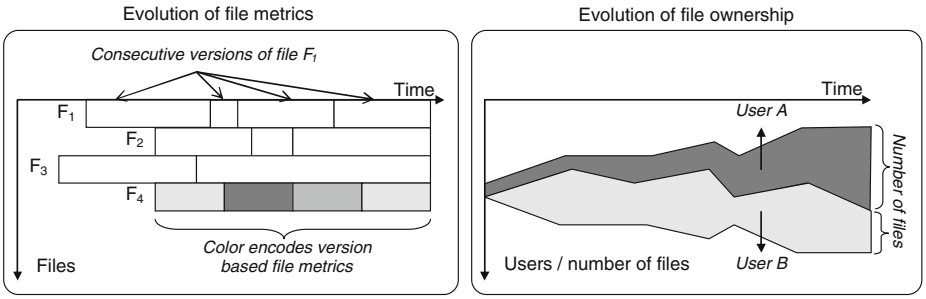

Fig. 2 CVSgrab visualization of project evolution

Summarizing, the main advantages of our framework as compared to the various repository data mining and/or visualization frameworks, are

- A dataflow architecture combining data acquisition, filtering, metric computation, storage, and visualization modules;
- Integrated support for CVS and Subversion repositories;
- Handling specific CVS data formats by scripted plug-ins;
- Incremental and cached access to data on repositories;
- An integrated module for clustering files based on evolutionary coupling;
- An integrated visualization module in which all visualization options (e.g. layout, color encoding, textures) can be customized to reflect any data attribute value (e.g. file size, type, date, author).

In the remainder of this article, we present several case studies that illustrate the applicability of our framework on real-life, industry-size software repositories. In all these studies, we connected the evolutionary coupling and clustering algorithm (Section 3.2) to the CVSgrab visualization back-end (outlined above) through the data acquisition mediator (Section 3.1) to form a complete evolution analysis solution. This solution was then used by us to answer the questions stated in the introduction. Next, we confronted domain experts with the results and other existing information in order to (in)validate them. Finally, we presented the complete toolset to the domain experts and collected feedback on other issues, such as usability and intuitiveness.

4 Case Study: Assessment of Repository Management Style

In the first study, we analyzed the evolution of two open source software (OSS) projects to investigate how changes are committed in repositories. In order to ensure consistency in OSS projects where many developers join the team, only a few developers are typically allowed to make project-wide changes. Often, the changes committed by these 'repository managers' incorporate the work of many other developers who do not have commit rights, represent project-wide updates such

as code beautification, or represent considerable work gathered into a single 'bulk commit'. Hence, simply equating the user(s) who perform the most changes with the most insightful persons in the project can be misleading. Those users might have committed the changes of other less privileged developers. Also, code beautification operations touch a lot of files, but that does not mean the user performing them has intimate knowledge over them.

To assess if the above takes place (or not) for a given repository, i.e. to answer questions $Q1$ and $Q2$, we performed an investigation using the framework introduced in Section 3 and validated the results using the information available in the corresponding project documentation and on the project website. We examined two projects: ArgoUML and PostgreSQL.

4.1 ArgoUML

The first project we analyzed was ArgoUML, an object-oriented design tool with a 6-year evolution of 4,452 files developed by 37 authors. Using the data acquisition mediator (Section 3.1), we retrieved the project information from the public CVS repository. For this, we had to extend the CVS format converter such that file names with spaces were correctly handled.

Once the format converter had been patched, data acquisition took 31 min over a T1 Internet connection: 8 min for the initial setup (i.e., retrieval, in full, of the last version of 59 MB) and 23 min to retrieve the evolution data to be visualized (20 MB). After being retrieved, all data was cached by the acquisition mediator. Any subsequent updates of this information, e.g. when new versions appear in the repository, require only a time proportional with the number of changed files. Hence, when only a few files are changed, updating the cached data only requires a couple of seconds.

After acquiring the evolution data with the mediator, we visualized it using CVSgrab. Figure 3 presents several images depicting the evolution of ArgoUML.

In Fig. 3a, the 4,452 files are sorted along the y axis in decreasing order of the creation time: old files are at the bottom, younger files are at the top. Gray shade indicates the file type: light gray is for documentation files, dark gray is for implementation (source code) files.[2] We see that, in the beginning of the project, many documentation files (roughly 400) have been committed, apparently during one transaction, as indicated by a steep vertical pattern in the image. A similar 'bulk commit' of implementation files followed a few months later. In Fig. 3b, files are similarly arranged, but color encodes the ID of the developer who committed each file. We see that the two chunks of files committed in bulk in the beginning of the project, but also other similar chunks committed later, are contributed by individual developers. The first chunk (documentation) has been committed by the developer *jrobbis*. Many files committed in bulk in one transaction by a single person suggests that previous development existed, and/or that cumulated work of more developers has been committed by a code manager.

[2]The CVSgrab tool produces full-color visualizations. These have been converted to grayscale for printing purposes.

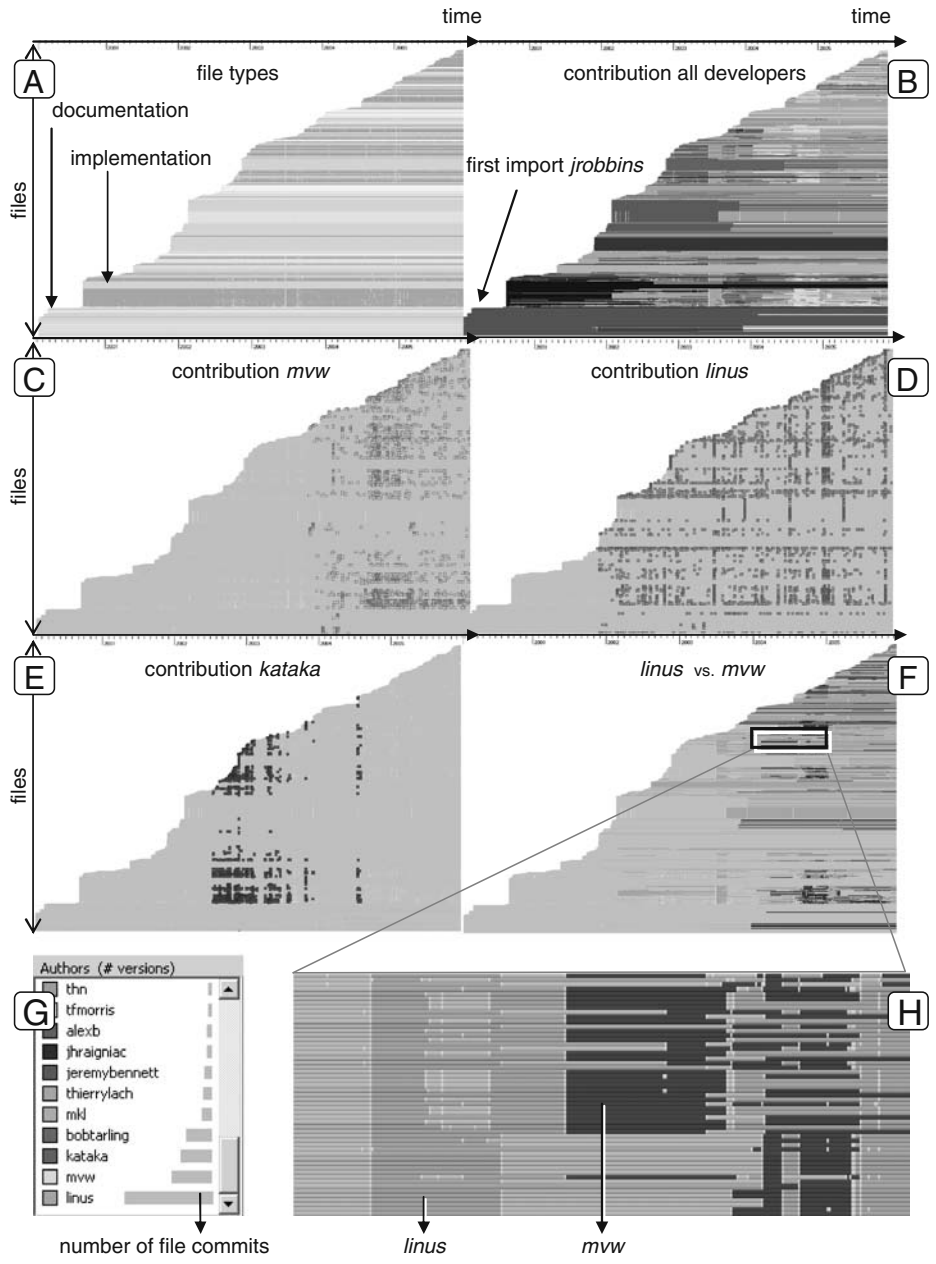

Fig. 3 ArgoUML evolution visualization with CVSgrab

Figure 3g shows a list of the most active developers. The list is sorted based on the number of file commits associated with each person, shown as a bar for each person name. We see that *linus*, *mvw* and *kataka* are the most active developers. However, the many files they contribute (compared to other developers) suggests they have the

role of code managers, i.e. they commit the work of more developers, taking care of the code consistency.

Figure 3e depicts the commit moments of *kataka*. Dark gray dots show the file versions he committed. We see that *kataka* has not been active in the last 18 months. In contrast, Fig. 3c and d, indicates that *linus* and *mvw* are active code managers.

Figure 3f uses color to visualize the contributions of *linus* (light gray) and *mvw* (dark gray). Figure 3h shows a zoom-in on a selected area. In both images, we see that both *linus* and *mvw* have a similar contribution style. They typically commit many files at one moment, but also make individual smaller commits. This suggests they both do project-wide operations, such as code beautification, and also do issue-specific code updates, possibly provided by other developers.

We validated our observations on the evolution of ArgoUML by comparing them with the information available in the project documentation and on the official project website (ArgoUML, www.http://argouml.tigris.org/). According to this information, ArgoUML has been started by Jason Robbins prior to 1995, and preliminary versions have been made public as early as 1998. The repository has been only populated in early 2000, by the same developer. According to the same website, prior to 1999, more developers took part in the development of ArgoUML, for example Jim Holt and Curt Arnold. This validates our hypothesis that previous work, done by several developers, existed before the first files were stored in the repository.

We also consulted the list of committed members available on the project website to investigate the role of *kataka*, *linus* and *mvw*. According to the list, *kataka* has been a major developer only until release 0.18 of ArgoUML, in April 2005, and did not contribute to later releases. This confirmed our observation about *kataka* being an important developer who was inactive in the last 18 months of recorded development. The list further revealed that *linus* has the role of a project leader, being also responsible for all releases. This explains his contribution pattern, which affects in general many files at the same moment, and confirms our hypothesis of *linus* being an important code manager. Additionally, the list reveals that *mvw* is in charge of code documentation. This can also explain why he modifies many files at once, performing project-wide modifications that may not be issue-specific.

4.2 PostgreSQL

The second project we investigated was PostgreSQL, an object-relational database management system with a history of 10 years, 2,829 files, and 27 authors. We followed a similar procedure as in the first case. We started with the data acquisition step. Our CVS format converter needed no adaptation for this project. The acquisition took 28 min: 7 min for the initial setup (i.e., retrieval of the last project version = 56 MB) and 21 min for retrieving the evolution data to be visualized (29 MB). The evolution retrieval time was smaller than in our first example, even though more data was retrieved. This is explained by the connection overhead. In CVS, when retrieving evolution data, the connection has to be established for each file. PostgreSQL has less files than ArgoUML, which decreased the overall connection latency.

After acquisition, we used CVSgrab to visualize the data.[3] Figure 4 shows several generated images depicting the evolution of PostgreSQL.

Figure 4a shows the project's 2,829 files using a layout sorted by creation date, similar to the previously presented case. Color shows file type: implementation files are dark gray, headers are light gray. Similar to ArgoUML, we see that both implementation and header files have been committed initially in large chunks during bulk transactions, as shown by those steep vertical jumps in the image. Additionally, Fig. 4b shows that all files in the two initial chunks are committed by only one person: *scrappy*. Whereas these findings may not be immediately visible on the overview images shown in Fig. 4, they are quite easy to see in the actual tool, in which each image is viewed at full-screen resolution and can be further zoomed in, if the commit density or file count are too high.

Figure 4g shows a list of the most active developers. According to it, *momjian* and *tgl* are the most active ones. Figure 4c and d also shows that they are still active, i.e. committed recently.

Figure 4f uses color to visualize the contributions of *momjian* (light gray) and *tgl* (black), with a zoom-in shown in Fig. 4h. These images show that these developers have very different contribution styles. While *momjian* changes many files in single transactions, *tgl* commits only a few files at once, but does many transactions. Based on similar patterns that we have seen in several other OSS repositories, this suggests that *momjian* may be involved in project-wide code updates (e.g. code beautification, patches) while *tgl* may commit issue-specific code updates from other developers.

We validated our observations on the evolution of PostgreSQL by confronting them with the facts from the official project website (http://www.postgresql.org/). According to these, the project roots go back to 1986. It was developed by many teams along the time, yet the first files were stored in the repository only in the first half of 1996 by Marc G. Fournier. This developer has the ID *scrappy* and one of his roles is to manage the repository. This confirms our hypothesis that previous work existed before the first commit and that these files were the work of more developers.

According to the website, *momjian* (Bruce Momjian) and *tgl* (Tom Lane) are two of the core developers behind the project. Besides being a core developer *momjian*, has also the task of applying patches contributed by other developers. Also, *tgl* is presented as being involved all aspects of PostgreSQL, including bug evaluation and fixes, performance improvements, and major new features. The above confirm both our hypotheses regarding *momjian* and *tgl*.

Concluding, the two case studies on the evolution of ArgoUML and PostgreSQL show that commit data does, in some cases, represent the activity of a 'code manager' and not the activity of all developers involved in a project. On the one hand, previous development efforts may have existed before the repository was first populated. On the other hand, in order to ensure code consistency, developers can be constrained to commit their work via 'code managers'. Both cases can be detected by studying patterns in the repository evolution.

[3]The mediator makes it possible to couple CVSgrab visualizations with both CVS and Subversion repository data.

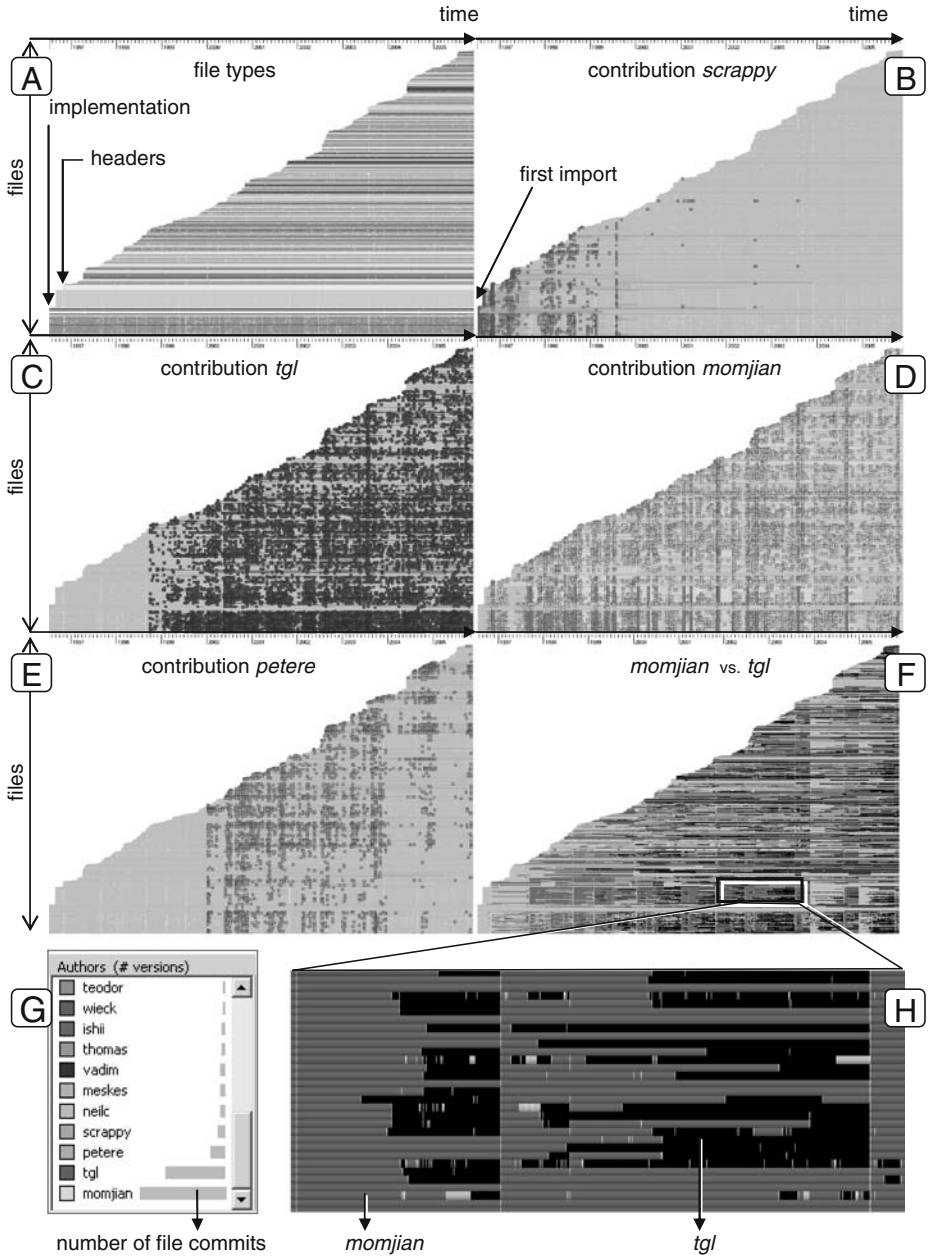

Fig. 4 PostgreSQL evolution visualization with CVSgrab

5 Case Study: System Decomposition Based of Evolutionary Coupling

An important question that most developers ask when trying to understand a software system is "what are the high-level building blocks?" ($Q3$). To answer this question, we use the history information stored in SCM repositories, by computing a system decomposition hierarchy based on evolutionary coupling (Section 3.2). However technically simple to compute, one should still ask: How accurately does such a decomposition reflect the actual building blocks used in the software?

We answered this question by studying the decomposition based on evolutionary coupling of the mCRL2 software package (http://www.mcrl2.org). mCRL2 is a toolset for modeling, validation, and verification of concurrent systems and protocols. It contains 2,635 files contributed by 15 developers during 30 development months. mCRL2 is hosted on a Subversion repository.

We started with the data acquisition step. The Subversion format converter needed no adaptations for this project. The acquisition took 20 min. This relatively low acquisition time as compared with the previous cases is explained by the richer functionality of Subversion repositories, which can deliver a list of their contents. Consequently, the initial set-up step specific to CVS repositories was not required in this case.

Once the information has been acquired, we used the evolutionary coupling decomposition component of our framework (Section 3.2). The component operates on the evolution data cached by the data acquisition mediator. We planned to validate the results by using a domain expert involved in the development of mCRL2 to assess the computed decomposition. We used as domain expert Jeroen van der Wulp, one of the main architects behind mCRL2. At his suggestion, we applied the decomposition on the `sip` subsystem, i.e. that part of the software with which he was most familiar. This subsystem contains 65 files and implements the interface between various tools available in mCRL2, and a central controller. The computation of the agglomerative clustering tree took only 2 s.

After the clustering, we used CVSgrab to select a decomposition based on similar cohesion (Voinea and Telea 2006a). Figure 5 shows the resulting decomposition, containing six main clusters (numbered 1–6) and a number of unclustered files (A and C). Files in the same cluster are visually emphasized using the so-called *plateau cushions*, dark at the edges and bright in the middle, implemented as described in (Voinea and Telea 2006a). We concluded that the `sip` component has six high-level building blocks, and we asked the opinion of the domain expert on this decomposition.

The expert analyzed the names of the files in each cluster and then indicated the main role they refer to in the design of the `sip` component. He identified clusters 1, 2, 3, and 6 as a meaningful view on the architecture of the `sip` component with 4 building blocks. The meaning of each clusters is shown as annotations in Fig. 5. Clusters 1, 2, and 6 contain mostly files correctly located in their corresponding building block (hence they are a good decomposition). Cluster 3 contains also a few files logically belonging to other blocks (hence is a less useful decomposition). Cluster 5 does not contains files in a building block, but groups files used for testing the `sip` subsystem. Finally, cluster 4 groups the remainder of the files in the project. The domain expert analyzed also the names of the unclustered files (e.g. A,C in Fig. 5) and concluded they are weakly related to one of the main clusters 1, 2, 3, or 6. In the

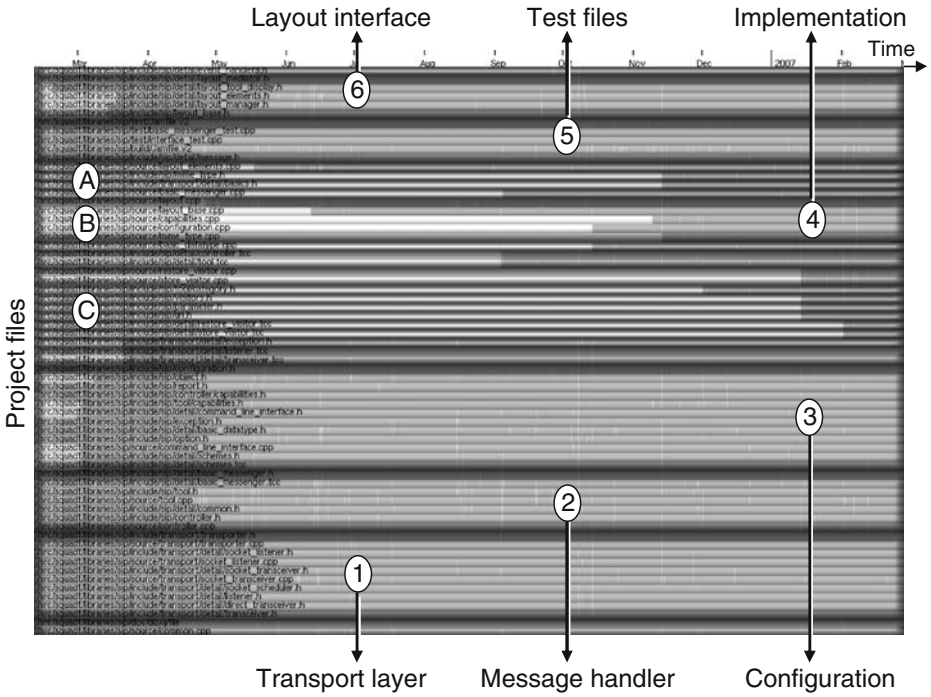

Fig. 5 *mCRL2* decomposition in building blocks based on evolutionary coupling

end, the domain expert considered that, with the exception of cluster 4, the evolution-based decomposition correctly revealed the most relevant building blocks in `sip`.

Our conclusion is that a system decomposition purely based on evolutionary coupling (i.e. without examining the file contents) is meaningful and useful. Yet, several aspects must be considered in practice. First, the decomposition covers all development activities that commit files. Hence, not only source code clusters are produced, but also clusters related to other activities such as testing. Secondly, artificial clusters will typically appear. In our study, cluster 4 does not have any special meaning. At a close analysis, we found out that cluster 4 contained files which were modified together during a project-wide activity (reconfiguration of the documentation engine). To eliminate such clusters, one should adapt the similarity metric (1) to include not just raw commit times, but additional semantics, such as file types, folder locations, or author IDs.

6 Case Study: Assessment of Maintainability

A frequent question that appears when assessing a third-party software stack for the first time is: "How maintainable is this software?" (*Q*4). Assessing past maintainability can help predicting future maintainability and is also to be considered when buying or outsourcing software stacks. Two important project history-related factors are useful here (among many others): The *development stability*, i.e., how

dynamically is the code changing, and the *code cleanness*, i.e., what is the quality of the intermediate versions.

We chose the KDE Pim project (Pim KDE, http://pim.kde.org) as a case study on which to answer these questions. KDE Pim is a sub-project of the KDE desktop environment. It offers an application suite to manage personal information. The project contains 6,141 files, contributed by 73 developers during almost eight development years. We used the Subversion module of the data acquisition mediator to retrieve the entire KDE Pim project history. This operation took 30 min. After acquisition, we used the CVSgrab visualization module to display the project evolution.

To validate the results of our investigation, we asked a domain expert involved in the development of KDE Pim to assess our findings. Our domain expert was Adriaan de Groot, one of the core KDE developers. We examined that part of the software that was most familiar to this domain expert, namely the KPilot application of the KDE Pim suite. KPilot contains 504 files. We used CVSgrab's filter functions to select the 145 C++ source files for detailed investigation. Next, we used a line counter plug-in component to add a lines-of-code-per-file (LOC) metric to the database maintained by the mediator.

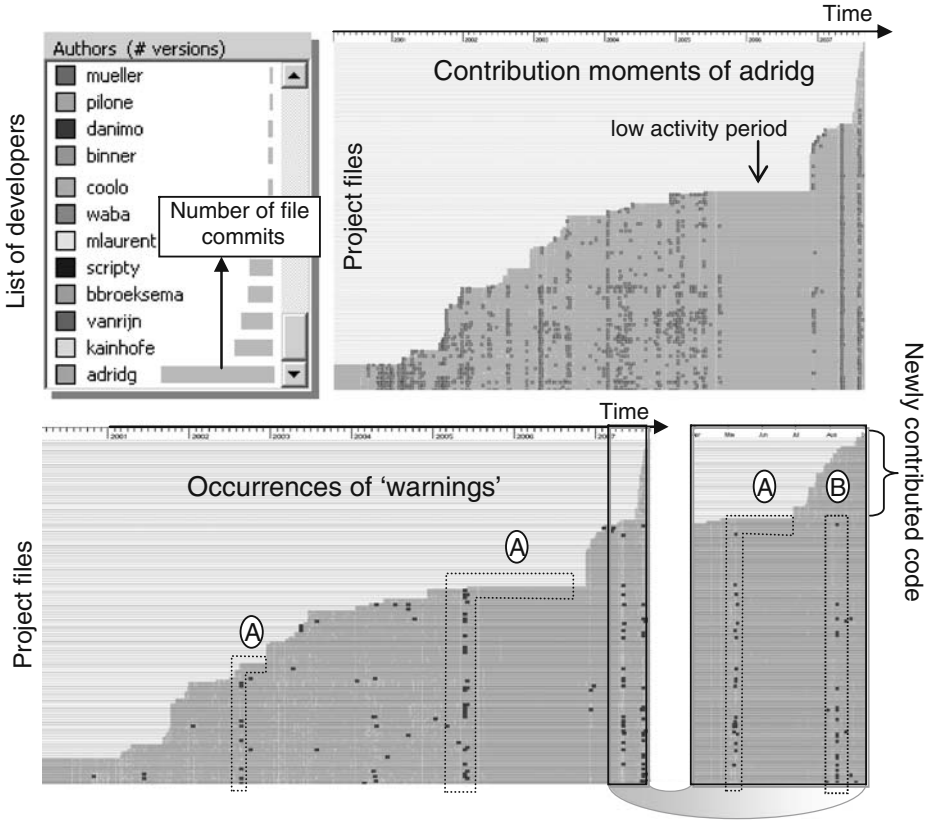

Fig. 6 Source code evolution in KDE Koffice. Code cleaning patterns, occurring at regular intervals, are emphasized (bottom images)

Figure 6 (top left) shows the developers sorted on number of commits, which yields *adridg* (Adriaan de Groot) as the main developer. In Fig. 6 (top right), files are drawn sorted on inverse creation time—old files are at bottom, most recent ones are at top. The dark dots show the commits (time and file) of *adridg*. We quickly see a flat region without any commit events (dark dots). This is a low-activity development moment, in which we can assume the software is in a stable state. Another interesting aspect regarding the maintainability of KPilot is shown in Fig. 6 (bottom). Files are laid out as above, but the dots mark the presence of the word "warnings" in the commit logs. This word often appears in logs reporting that compiler warnings have been removed, indicating that a thorough code inspection has been done. Such code cleaning happens when the code is stable and no planned (refactoring) changes are foreseen. In Fig. 6 (bottom), we see several "L" shaped patterns (A) that indicate many files being changed while removing "warnings", right before a low activity period. This suggests that as functionality is implemented and the software has reached a stable phase, the code is carefully inspected and compiler warnings are removed before continuing development. This suggests KPilot is a thoroughly maintained project. A different project-wide code clean step is seen in the zoom-in image (marker B). Here, only old code is modified to remove compile warnings.

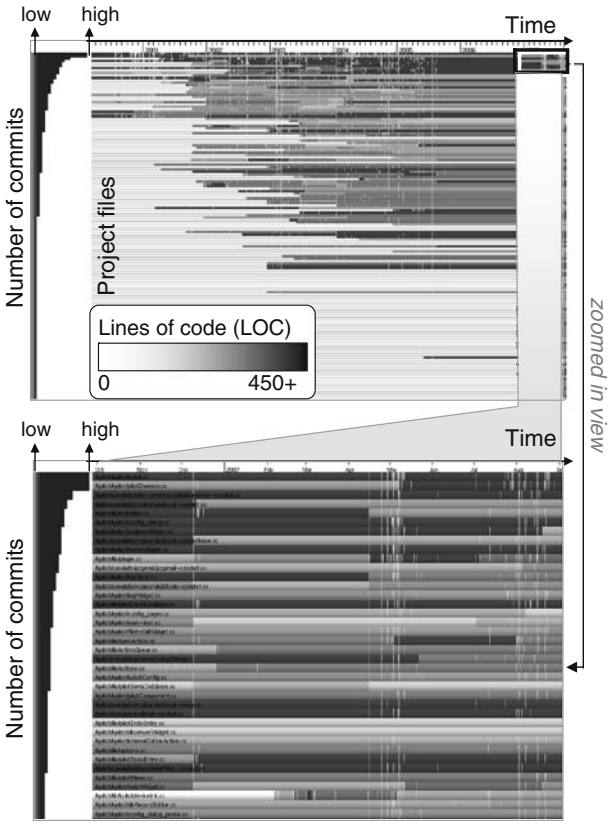

Fig. 7 Source code evolution in KDE Koffice. Color shows the lines-of-code per file variation in time. This allows detecting unstable, heavily changing, files

The newly contributed code (at top of image) is probably not yet stable, so it is not cleaned up.

Figure 7 shows code files sorted on activity (number of commits) and colored with the LOC metric. As expected, the most active files (at top) are also among the oldest. Since we measure activity as absolute number of commits, older files which are still changed get potentially a higher chance to be recognized as active. To emphasize recently active files, if desired, we can use the activity metric computed on a subperiod, e.g. few latest months, of the entire history.

However, we also discover these most active files are not very stable. The magnified region shows the latest versions of some of the most active files. These are not stable files, as the LOC metric (gray shade) varies a lot for each file.

Our findings described above were validated by the domain expert, who could recognize his work patterns in our results. We conclude that the visual correlation of several attributes, such as project activity, specific patterns in commit logs, and source code metrics, can deliver valuable insights on code stability and maintenance style (e.g. periodic code cleaning operations). Hence, the presence of such patterns can be used as a sign of good maintainability. However, as it appeared when asking a developer unfamiliar with the KPilot code base and only slightly familiar with CVSgrab, identifying such visual patterns clearly takes some training time (over one hour). An improvement we consider is to detect and visually highlight patterns of interest.

7 Case Study: Assessment of Project Team Risks

A set of important questions for project managers are "What are the team-related risks?", "What happens if a developers leaves the team?" and "What will be the effort of replacing the leaving person?" ($Q5$) In this section, we show how evolution data, mined from software repositories, can help answering these questions.

The first project we investigated was mCRL2 (http://www.mcrl2.org), same as in our second case study. As the evolution data was previously cached by the mediator (Section 5), this study started with no acquisition time. Next, we visualized the evolution of mCRL2 at system level to assess the distribution of impact in the team. We used a so-called *flow graph* to compare the number of files each developer "owned", i.e., which were last committed by that developer (see Fig. 8). For each developer, a tube-like band is drawn whose thickness (height) is scaled by the number of files owned. Time is mapped to the x axis, so the thickness variation shows the impact variation of a developer in the project. This gives an indication about which users are familiar with the repository at a given moment in time. Since this visualization shows changes cumulated per version, small-scale changes done sporadically by a user to a few files in a few versions will not considerably influence its ownership percentage in general. If desired, we can also compute a more fine-grained ownership per line-of-code, as shown in (Voinea et al. 2005), and aggregate that instead the coarser per-file ownership.

Figure 8 top shows the flow graph visualization for the mCRL2 repository. The small upper-left window shows the developers, sorted in increasing order of commit numbers, as described in the previous case studies (Figs. 3, 4, 7). In the second half of the first development year, an important developer (*jwulp*) joined the team.

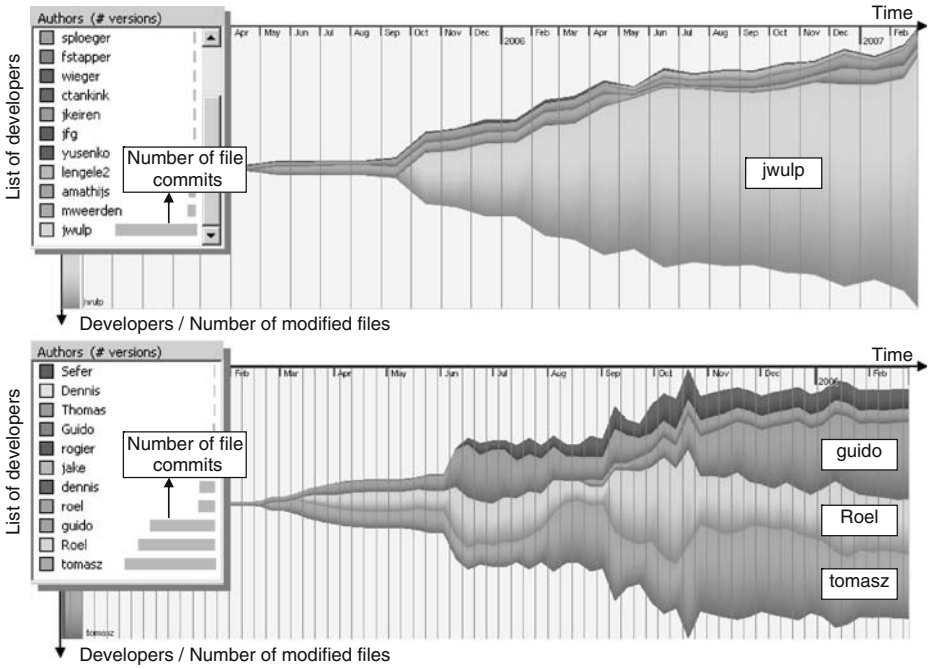

Fig. 8 Assessment of knowledge distribution in mCRL2 (*top*) and MagnaView (*bottom*)

Gradually, he became the owner of most implemented code. Given the impact distribution over the developers, he is arguably the main (and only) one up-to-date with changes in most parts of the project. From this perspective, he represents an important asset for the project, but also a high risk. If he left the team, a high amount of knowledge would have to be transferred to other team members. Our finding was validated by the main architect of mCRL2, who happens to be exactly this person, and was also confirmed by his project manager.

The second project we investigated is MagnaView (http://www.magnaview.nl), a commercial visualization software package containing 312 files, written by 11 developers in over 16 development months. Figure 8 bottom shows the flow graph visualization for this project. We see that in the first three months the relatively small code was owned by just one developer. At the end of February 2005, two new developers joined the team: *tomasz* and *Roel*. Developers *tomasz* and *Roel* seemed to own equal code amounts in the following months, while other minor developers have little influence. Starting June 2005, the code size increases notably. End of August 2005, a new developer (*guido*) joins the team. Until the end, the amount of code owned by *guido* increases steadily. *In the same time* the amount owned by the project initiator, *roel* (not to be confounded with *Roel*), shrinks significantly. At the end, the code is mainly owned by three developers: *tomasz*, *guido*, and *Roel*. We assumed these three developers have now a good understanding about the system and are important, and actually critical, team members. They own fairly balanced amounts of code, so the impact of losing one of them is quite small. The amount of code

owned by the other eight minor developers is very small, so they are non-critical to the development.

We used as validation the opinion of Roel Vliegen, the main software architect behind MagnaView, as domain expert. Our distribution of developers in terms of (critical) team impact was found to be correct. We concluded that the flow graph visualization showing the evolution of team members' impact provides a simple, intuitive, and highly effective way for managers to discover and/or monitor the impact distribution over a team. The flow graphs were particularly appreciated exactly because of their simplicity, as compared to more complex imagery such as the decomposition clusters (Fig. 5). In particular, flow graphs were easily understood, and appreciated, by non-technical persons (managers). However, the flow graph view was most effective when correlated with other views, such as the member list sorted based on impact, as well as the evolution of file metrics (Fig. 2 left).

8 Discussion

In the previous sections, we have described a number of case studies showing how our visualization framework can be used to discover several facts and perform several assessments on software repositories. At this point, an important question is: How much can we generalize from these studies to the applicability of our approach and the presented analyses to *any* software repository in general?

Several observations can be made here. First, all the chosen repositories (ArgoUML, PostgreSQL, mCRL2, KDE Pim, and MagnaView) contain large projects spanning several thousands of files each, tens of developers, and changes covering several years. During related past studies, we also analyzed the Mozilla code, which contains over ten thousand files (Voinea and Telea 2007). This makes us believe that our methods and tools are technically *scalable* to industry-size repositories. Second, the techniques presented here do not analyze the file *contents*, but only the repository commit data. While this makes the range of performed analyses less specific, e.g. we could not perform duplication or clone evolution analysis on source code, this also makes our techniques lightweight, fast, and generic, as we do not need to transfer or analyze the actual file contents.

Second, we chose for the case studies repositories ranging from OSS projects (ArgoUML, PostgreSQL, KDE Pim) to academic software (mCRL2) and commercial software (MagnaView). Software is developed and maintained in different ways in these three types of communities. We performed our different types of analyses (repository management style, system decomposition using evolutionary coupling, maintainability assessment, and team risk assessment) on all these repositories, although we described here only a subset thereof, for space constraints. From these results, we believe that the proposed methods and techniques should be applicable to any type of software repository in a range of organizations, as long as the repository data is accessible via a suitable mediator.

Third, we chose different types of information for validation: project history manually extracted from documentation and websites (ArgoUML, PostgreSQL) and discussions with actual developers and project managers (mCRL2, KDE Pim, and MagnaView). In none of the case studies were we involved, directly or not, in the

development or code-level usage of the studied code bases. In two cases (mCRL2 and MagnaView), the domain experts involved in the validation briefly saw our visualizations, but they were not told of our interpretation of the visual patterns until after they validated our findings. Given these, we conclude that users familiar with our visualization tools can effectively perform correct assessments of several evolution aspects on unknown repositories.

A separate question to discuss relates to the *effectiveness* of our visualizations. A visualization is effective if users can detect patterns in it which correlate with events in the displayed data. So which are the most salient patterns of our evolution visualizations, and why do they show?

One of the most frequently used visual patterns are *colored pixel blocks*. Given the 2D time-by-file layout used (Fig. 2), semi-compact, same-color, near-rectangular pixel blocks show file versions which are related by two attributes (sorting (y axis) and coloring) and related in time (x axis). After such blocks are found, their meaning depends on the axes' and color encoding, showing e.g. owned code (Section 4) or code-cleaning events or unstable files (Section 6). Clusters essentially use the same rectangular visual pattern, on a higher scape, to show this time files evolving together (Section 5). A quite different pattern are the 'code flows' used to show evolution of developers' impact (Section 7). This pattern is easy to interpret: tube thickness shows contribution amount, color shows identity. Thickening tubes, from right to left, show 'code owners' who become gradually more important. A very similar pattern was successfully used in computer games to show evolution of players in time (Microsoft Inc 2007).

9 Conclusions

We presented a framework for visual data mining and analysis of software repositories and its application in supporting various assessments of software development and maintenance processes. Our main goals were to determine, through case studies with concrete projects, developers, and questions, whether the insights generated by our tools do indeed match the reality; which were the most usable and useful tools; and to test the effectiveness of visual analysis tools for understanding software repositories in practice.

Through this study, we have reached several conclusions. The critical requirements for visual analysis tools (on software repositories) are *simplicity*, *genericity*, and *integration*. Visual techniques (and their user interfaces) need to be *simple* to be understood and accepted. Our simplest techniques: 2D Cartesian layouts, color encoding attributes, and sorting on any attribute, are easy to grasp and perform interactively with a few mouse clicks. We presented also a new visualization, the flow graphs, which shows team impact distribution over time in a simple, aggregated, manner. This visualization was especially appreciated by project managers who are typically farther away from the code. More complex techniques, such as the evolutionary coupling based clusters (Section 3.2, are considerably harder to understand, even by more advanced users. Next, the proposed techniques have to be *generic*: any technique (e.g. color mapping or sorting) can be applied on any attribute (e.g. author IDs or the LOC metric). This allows users to construct analysis scenarios

interactively with a few mouse clicks. Finally, the complete framework has to be fully *integrated* with existing repositories. Our solution, a network of scripted components connected by a dataflow engine to a central fact database, allows rapid customization for different repository standards and formats. This proved to be essential in doing the case studies, as all users we talked to, whether from the academia or the industry, suggested that they would generally reject a tool-chain having a complex, tedious set-up phase.

Our second goal was to develop an experimenting framework, both in software tools and methodology, for research on software evolution. Our framework, consisting of a set of data filters, metric computation components, and a visualization back-end, all coupled together via a scriptable data acquisition mediator to the actual repositories, provides a flexible way to construct visual analysis scenarios targeting concrete questions in a few minutes up to about one hour. Our framework is illustrated by four different case studies (of a larger set that we performed), covering analysis of repository management style, system decomposition, maintainability, and project team risks, over three industry-size Open Source projects (ArgoUML, PostgreSQL, KDE Koffice), one academic system (mCRL2) and one commercial system (MagnaView). Virtually all our findings were positively verified by domain experts who were not involved in the analysis. Also, we had no prior knowledge about the analyzed systems.

This positive news has to be, however, put into perspective. In all cases, the persons performing the visual analysis were reasonably familiar with our visualization framework. An important subsequent question is whether actual developers in the field will be able to get the same results with our tools in a comparable amount of time. Answering this is still an open question. In a follow-up set of case studies, we plan to assess the usability of our toolset by its end-users and, also, refine and extend those visual techniques which are best accepted by the users. Finally, we are working on extending our toolset with new metrics (e.g. an improved evolutionary coupling) that target new specific questions of our users.

References

Ball T, Kim JM, Porter AA, Siy HP (1997) If your version control system could talk.... In: Proc. ICSE '97 workshop on process modeling and empirical studies of software engineering

Bennett K, Burd E, Kemerer C, Lehman MM, Lee M, Madachy R, Mair C, Sjoberg D, Slaughter S (1999) Empirical studies of evolving systems. Empirical Soft Eng 4(4):370–380

Bieman JM, Andrews AA, Yang HJ (2003) Understanding change-proneness in oo software through visualization. In: IWPC'03: Proc. intl. workshop on program comprehension. IEEE CS Press, pp 44–53

Burch M, Diehl S, Weißgerber P (2005) Visual data mining in software archives. In: SoftVis '05: Proc. ACM symposium on software visualization. ACM Press, pp 37–46

Collberg C, Kobourov S, Nagra J, Pitts J, Wampler K (2003) A system for graph-based visualization of the evolution of software. In: SoftVis'03: Proc. ACM symposium on software visualization. ACM Press, pp 77–86

Cubranic D, Murphy GC, Singer J, Booth KS (2005) Hipikat: a project memory for software development. IEEE Trans Softw Eng 31(6):446–465

Eick SG, Steffen JL, Sumner EE (1992) SeeSoft—a tool for visualizing line oriented software statistics. IEEE Trans Soft Eng 18(11):957–968

Everitt E, Landau S, Leese M (2001) Cluster analysis. Arnold Publishers, Inc
Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: ICSM'03: Proc. intl. conference on software maintenance. IEEE CS Press, pp 23–32
Froehlich J, Dourish P (2004) Unifying artifacts and activities in a visual tool for distributed software development teams. In: ICSE'04: Proc. intl. conference on software engineering. IEEE CS Press, pp 387–396
Gall H, Jazayeri M, Krajewski J (2003) CVS release history data for detecting logical couplings. In: IWPSE'03: Proc. intl. workshop on principles of software evolution. IEEE CS Press, pp 13–23
German D, Mockus A (2003) Automating the measurement of open source projects. In: Proc. ICSE'03 workshop on open source software engineering, pp 63–38
German D, Hindle A, Jordan N (2004) Visualizing the evolution of software using SoftChange. In: ICSEKE'04: Proc. 16th intl. conference on software engineering and knowledge engineering, pp 336–341
Greenwood RM, Warboys B, Harrison R, Henderson P (1998) An empirical study of the evolution of a software system. In: ASE'98: Proc. 13^{th} conference on automated software engineering. IEEE CS Press, pp 293–296
Lanza M (2001) The evolution matrix: recovering software evolution using software visualization techniques. In: IWPSE'01: Proc. intl. workshop on principles of software evolution. ACM Press, pp 37–42
Lopez-Fernandez L, Robles G, Gonzalez-Barahona JM (2004) Applying social network analysis to the information in cvs repositories. In: MSR'04: Proc. intl. workshop on mining software repositories. IEEE CS Press
Microsoft Inc (2007) Age of empires game. www.microsoft.com/games/empires
Voinea L, Telea A (2006a) CVSgrab: mining the history of large software projects. In: EuroVis'06: Proc. eurographics/IEEE-VGTC symposium on visualization. IEEE CS Press, pp 187–194
Voinea L, Telea A (2006b) How do changes in buggy Mozilla files propagate? In: SoftVis '06: Proc. ACM symposium on software visualization. ACM Press, pp 147–148
Voinea L, Telea A (2006c) Mining software repositories with CVSgrab. In: MSR '06: Proc. intl. workshop on mining software repositories. ACM Press, pp 167–168
Voinea L, Telea A (2006d) Multiscale and multivariate visualizations of software evolution. In: SoftVis '06: Proceedings of the 2006 ACM symposium on software visualization. ACM Press, pp 115–124
Voinea L, Telea A (2007) Visual data mining and analysis of software repositories. Comput Graph 31(3):410–428
Voinea L, Telea A, van Wijk JJ (2005) Visualization of code evolution. In: SoftVis'05: Proc. ACM symposium on software visualization. ACM Press, pp 47–56
Wu J, Spitzer C, Hassan A, Holt R (2004a) Evolution spectrographs: visualizing punctuated change in software evolution. In: IWPSE'04: Proc. intl. workshop on principles of software evolution. IEEE CS Press, pp 57–66
Wu X, Murray A, Storey MA, Lintern R (2004b) A reverse engineering approach to support software maintenance: version control knowledge extraction. In: WCRE '04: Proceedings of the 11th working conference on reverse engineering (WCRE'04). IEEE Computer Society, Washington, DC, USA, pp 90–99
Ying ATT, Murphy GC, Ng R, Chu-Carroll MC (2004) Predicting source code changes by mining revision history. IEEE Trans Soft Eng 30(9):574–586
Zimmermann T, Weisgerber P (2004) Preprocessing *CVS* data for fine-grained analysis. In: MSR'04: Proc. intl. workshop on mining software repositories
Zimmermann T, Weisgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: ICSE '04: Proc. intl. conference on software engineering. IEEE CS Press, pp 563–572

Lucian Voinea received a Professional Doctorate in Engineering degree (PDEng) from the Eindhoven University of Technology (Netherlands) in 2003 and a PhD degree in computer science from the same university in 2007. Starting from 1999, he worked as a freelance contractor for companies in Romania, Netherlands and US. His research interests include methods, technologies and tools for the analysis of quality attributes of large software systems, and in particular the analysis of software evolution. He recently co-founded SolidSource, a start-up company specialized in tools and services for the maintenance of software systems (www.solidsource.nl).

Alexandru Telea received his PhD in 2000 from the Eindhoven University of Technology in the Netherlands. He worked at the same university as an assistant professor in data visualization until 2007, when he received an adjunct professor position in software visualization from the University of Groningen, the Netherlands. He has pioneered several innovative methods in visualizing complex information related to software systems, reverse engineering, and software evolution. He has been the lead developer and architect of several software systems for reverse engineering, data visualization, visual programming, and component-based development. He has published over 100 articles and one book in the above fields.